Amendments to the Specification:

Please replace paragraph [0012] with the following rewritten paragraph:

[0012] Another serialization format is [[in]] the "Storage Engine record" format, also referred to as the "SE record," or simply "record" format. This is [[an]] a typical database system record format. In this serialization format, members for objects of a given class are stored in uniformly formatted records. Instead of providing metadata that describes each and every member, there is metadata that describes the contents of all the records for objects of a particular class. This can be visualized as provided in Fig. 10.

Please replace paragraph [0014] with the following rewritten paragraph:

[0014] Further[[ ]], various storage formats have been designed to allow users of databases to efficiently store objects within a database. These storage formats can be better supported with a more flexible serialization format. ~~For example, should be distinguished from the serialization format provided herein.~~ For example United States Patent Application No. 10/692,225, Attorney Docket No. MSFT 2852/306819.01, titled "system and method for object persistence in a database store," is directed to allowing a user to 'import' classes and methods written in an object oriented language like C# into a database. It further allows a user to store C# objects in a database and to invoke methods on the objects. It provides multiple flavors of persistence to a user. A user can define his own serialization format, use Common Language Runtime ("CLR") serialization (provided by C# language itself), or let the SQL server store an object in its own format. These options, particularly the latter, provide a performance advantage, as MICROSOFT SQL SERVER® can retrieve or update some fields of an object without actually instantiating a C# object. Of course, some operations, such as method invocation, still require instantiation of a C# object.

Please replace paragraph [0040] with the following rewritten paragraph:

[0040] **Binary Fragment** – Fig. 11(A) displays various potential embodiments of a Binary Fragment. This fragment can contain a Type, Length, and Payload field. The ~~type~~ Type field may be just one byte, or it can be any number of bytes. Additional bytes in the Type

field will require additional memory overhead when using the serialization format. Therefore bytes in header fields should be used sparingly. In this regard, a one-byte Type field can include a number of bits for use in indicating various properties of a fragment. One bit may be used to indicate that a fragment is a Binary Fragment. Another bit may be used to indicate a type of member or members contained in the fragment. For example, if all members are primitive, a bit may be set to indicate such information. If the members are subtype members, a bit may be set to so indicate. If the Binary Fragment is the first, or the only, fragment for a serialized object, [[A]] a bit in the Type field may so indicate. The type field may also indicate an object type contained in a fragment or fragments, as well as any additional useful information such as the number and types of fragments in the entire object.

Please replace paragraph [0048] with the following rewritten paragraph:

[0048] Terminator Fragment – Fig. 11(D) shows various possible embodiments for a Terminator Fragment. In the preferred implementation of fragment-based serialization, only the Type byte is relevant for the Terminator Fragment. This is because the function of the Terminator Fragment is to mark the end of a serialized object, or to mark the end of a collection or other set of related fragments within a serialized object. The Terminator Fragment can perform this function with a Type ~~filed~~ field indicating that it is a Terminator Fragment, and need not include additional information. However, it may be useful to include some additional information with the Terminator Fragment, and such embodiments are certainly within the scope of the invention described herein.